View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Electronic Communications of the EASST (European Association of Software Science and...

Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2

Jeroen J. A. Keiren and Martijn D. Klabbers

16 pages

Guest Editors: Gerald Lüttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2

Jeroen J. A. Keiren[1] **and Martijn D. Klabbers**[2]

Department of Mathematics & Computer Science, Eindhoven University of Technology[1]
LaQuSo, Laboratory for Quality Software, Eindhoven University of Technology[2]

Abstract: In this paper we advocate that formal verification should be a part of the development of a communication standard; in a short period of time issues are uncovered that have been in the standard for a number of years, and all subtleties in the correctness of the protocol are understood.

We model and verify the session setup protocol that is part of the IEEE 11073-20601:2008 standard for communication between personal health devices. We identify a number of issues present in the standards document. Discussion with a member of the standards committee unveiled that most, but not all, of the identified issues are fixed in the IEEE 11073-20601:2010 version of the standard. In addition, the correctness of the protocol, including the fixes, is assessed. For this, properties of the session setup protocol are formulated, and using the model checker mCRL2 it is verified whether the model satisfies these properties. We show that the session setup protocol is flawed, and propose a straightforward way to fix this issue.

Keywords: Verification, model checking, IEEE 11073-20601, protocol standardisation, interoperability

1 Introduction

Communication standards often prescribe an exchange protocol [MGWB03, DGRV00]. These protocols form the basis of development and testing of the communication of, sometimes critical, devices. Errors in these standards often lead to high costs in the development of such devices, and introduce safety threats in them [BB01]. Standards might even obstruct a solution for these errors, because a fixed version of the protocol does not conform to the standard any more, and may break backward compatibility.

IEEE 11073 forms a family of such standards for device communication, supporting a high level of automation—the user only needs to connect the devices, everything else is automatic. Within this family, the IEEE 11073-20601 standard focuses on *personal telehealth devices*. Examples of such devices are heart rate watches, glucose monitoring devices, and weighing scales.

IEEE 11073-20601 defines a generic data exchange protocol for these personal telehealth devices. It enables devices, called *agents*, to transfer patient specific health data to a *manager* with a low overhead. Devices allow for a large number of different *configurations*. In case of a weighing scale, the configuration can, *e.g.*, determine whether transferred weight data must be interpreted as kilogrammes or pounds.

Within the data exchange protocol described in IEEE 11073-20601 we can roughly identify two phases. In the *session setup* phase, an agent and a manager negotiate a configuration, in the second phase, measurement data is exchanged. The session setup protocol has to ensure that the agent and the manager are using the same configuration before measurement data is successfully communicated.

In this paper we only consider the session setup phase. This is motivated by the apparent importance of the session setup in the standard. The importance is indicated by the effort spent in describing session setup using a combination of UML state machines [Obj09], message sequence charts [Obj09], state transition tables and natural language. For other parts of the protocol one or more of these descriptions have been omitted.

The quality of the standard and the added value of formal techniques are assessed in the following three stages:

1. We assess understandability, consistency, and completeness of the 2008 version of the standard [IEE08] in Section 3. An initial model of this version of the standard is [IEE08] constructed in the language mCRL2 [GMR+08], which is an ACP [BK84] style process algebra with an accompanying verification toolset. During this first stage, we uncovered a number of issues in the standards document in a matter of hours. These issues have been present in the standard since 2008, and are are discussed in-depth in Section 3.2.

   At the time of the initial modelling a preliminary version of the 2010 update of the standard [IEE10] was under development, but not available to the authors of this paper. Discussion of the issues with a member of the standards committee showed that most, but not all, of the problems we discovered have been fixed in the 2010 version of the standard [IEE10]. As far as we are aware, we found all issues regarding the correctness of session setup that were fixed in the 2010 version of the standard. The issue that we found that has not been fixed is highlighted in Section 3.2.

   This initial part of our verification effort illustrates that building a formal model of a standard is an effective way of finding inconsistencies in the standard.

2. For verifying the correctness of the session setup protocol, we take the best known version of the standard. This means that the formal verification is carried out on the 2010 version of the standard, in which we have incorporated fixes for the problems that we observed in the first stage. We use the 2010 version instead of the 2008 version with our own fixes, since multiple fixes are correct for some of the problems that we found in that version; we would like our results to be accurate for the newest available version of the standard.

   The verification is described in Section 4, in which we first formulate a number of correctness requirements. The requirements for session-setup are not clearly documented in the standard, but the requirements that we verify have been validated with a member of the standards committee. The mCRL2 model is verified using modal μ-calculus formulae. We show that there is a subtle race condition in the protocol as it is described by the standard. This can lead to transmission of measurement data in a situation where an agent and manager are using different configurations.

3. In Section 5 we propose a straightforward fix for the the race condition that we found. We verify the fixed protocol using an ad-hoc abstraction, and show that the protocol guarantees that the agent and the manager use consistent configurations if measurements are transferred.

Our experiment confirms that formal modelling and verification form a useful addition to the standardisation process, even if the standard already employs some formal techniques for describing protocols. Protocol standards are susceptible to inconsistencies and incompleteness. Furthermore, unverified protocols are likely to contain subtle, hard to detect bugs, that are costly to fix once the standard has been released, if fixing is possible at all without breaking compliance of the currently approved devices with the new standard.

Related work Several medical protocol standards have been studied in the literature. Among others the ISO/IEEE 1073.2 standard for remote control [MG05, MGWB03]. Besides medical protocols, other standard protocols have been studied extensively. Examples are the IEEE 1394 firewire standard [DGRV00, Rom99, vRG03], contention resolution in the ZigBee protocol [Fru06], and the Carrier Sense Multiple Access/Collision Detection protocol in IEEE 802.3 [DFH+05].

Several case studies have been carried out showing the applicability of the model checker μCRL and its successor mCRL2. Among others an automated parking garage [MP07], a distributed system for lifting trucks [GPW03] as well as a printer intended to operate in the manufacturing of printed circuit boards [SR09] have been verified. The translation of UML state charts to mCRL2 that we use was described in [HKL+09].

Outline The rest of this paper is structured as follows. In Section 2 we provide some background on the formalisms background of the verification problem at hand, explain how this has been modelled in mCRL2, and discuss the quality of the standard. In Section 4 we verify the session setup protocol. The bug that we find is fixed in Section 5. Our conclusions and recommendations are presented in Section 6.

2 Preliminaries

Many real-life systems are concurrent. In a concurrent system, several processes run in parallel, either on the same hardware, or on different hardware. Processes communicate or interact with each other, and with their environment, to accomplish complex tasks. Communication between such processes can either be synchronous (sending and receiving happen at the same moment), or asynchronous (sending and receiving happen at different moments).

Various languages have been proposed for describing concurrent systems. These languages include state charts [Har87], Petri nets [Pet62], and process algebras [BBR10]. These languages can be used to describe the behaviour at a higher level of abstraction than programming languages. As such, the languages can be viewed as specification languages. Studying a system based on a high-level specification allows for analysing a *conceptual view* of the system, without having to cope with all implementation details.

The session setup phase of the protocol is described using state charts. For the analysis we use the specification language mCRL2 [GMR+08]. We briefly describe these formalisms and their translation in the rest of this section. The translation of the session setup is discussed in detail in Section 3.

State machines We assume some familiarity with UML state machines. For a detailed exposition of UML state machines we refer to [Obj09]. In this paper we use state machines that contain composite states (OR-states) and initial states. Transitions are labelled by a trigger and an action, both of which are optional. Triggers can be signals or change events. Signal events are communicated asynchronously, and can either be sent by the system, or by the environment. A change event is triggered when a certain condition becomes true. In Section 3 we introduce state machines in more detail, using part of the agent state machine as an example.

mCRL2 The language mCRL2 is based on the process algebra ACP [BK84], extended to include data and time. A fundamental concept in mCRL2 is the *process*. Processes can perform *actions*, and be combined into new process using algebraic operators. Systems usually consist of several processes in parallel. The language mCRL2 is supported by a tool set for the analysis of mCRL2 models.

Processes can carry data as parameters. The state of the process is a combination of values of the parameters. The actions that the process can perform may be influenced by this state, and the execution of actions may result in state changes. Every process has a corresponding state space, or Labelled Transition System, which contains all states that the process can reach, along with the transitions between the states.

A central notion in the verification process using mCRL2 is the linear process (LPS). An LPS represents a process in which all parallelism has been removed, resulting in a series of condition, action, effect rules. Model checking is performed using parameterised Boolean equation systems (PBES) [GW05]. Given an LPS and a formula expressing a requirement of the process, a PBES is generated. The solution to this PBES indicates whether the formula holds on the process. [1]

Translation A systematic translation of executable UML (xUML), which contains UML state machine as a subset, was described by Hansen *et al.* [HKL+09]. We use the translation described in *ibid.* to translate the state machines of the agent and manager processes to mCRL2. Note that we assume atomic run-to-completion, which means that, while a state machine is executing a local step, no event can be dispatched to any of the state machines in the system. A translation of part of the session setup protocol is provided in Section 3.

Modal μ-calculus We describe requirements in a variant of the modal μ-calculus, extended with regular expressions and data [GM99]. We use the following subset of this modal μ-calculus:

$$\begin{array}{lll} \varphi, \psi & ::= & b \mid \nu X.\varphi \mid \langle \rho \rangle \varphi \mid [\rho]\varphi \mid \varphi \wedge \psi \mid \neg \varphi \\ \rho & ::= & \alpha \mid \rho \cdot \rho \mid \rho^* \\ \alpha & ::= & \mathit{true} \mid a(\vec{d}) \mid \exists_{d:\,D} \alpha \mid \alpha \cup \alpha \mid \neg \alpha \end{array}$$

Here φ represents a property, ρ represents a set of sequences of actions, and α represents a set of multi-actions. The set of all multi-actions is denoted by *true*. Data can be introduced into a set of actions using existential quantification ($\exists$), sets of multi-actions can be combined using union ($\cup$), and complemented using negation ($\neg$). Sets of actions can be combined into action sequences using concatenation ($\cdot$) and iteration (*).

The property b is a Boolean expression that can include data expressions that occur free as parameters to actions. The Boolean formula *true* holds in every state. The property $[\rho]\varphi$ holds if φ holds in all states that can be reached by a sequence conforming to ρ. The property $\langle \rho \rangle \varphi$ holds if there exists such a reachable state. The greatest fixed point operator is ν. The property $\nu X.\langle a \rangle X$ holds in a state in which an infinite sequence of a actions is possible. Conjunction and negation of properties are as expected. For an in-depth description of the modal μ-calculus and its semantics, we refer to [GM99].

3 Session setup

The standard [IEE08] describes the communication between agents and a manager. Medical information is communicated from an agent to a manager in a reliable, asynchronous way. Various transport technologies, *e.g.* bluetooth, are supported.

Before any medical information, referred to as *data* in the rest of this paper, is transferred between the agent and the manager, both parties need to agree on a *configuration*. A configuration can be seen as an agreement on the meaning of the data that is transferred, *e.g.* the unit of weights is kilogrammes,

[1] For more information on mCRL2 see http://www.mcrl2.org. We used SVN revision 10606.

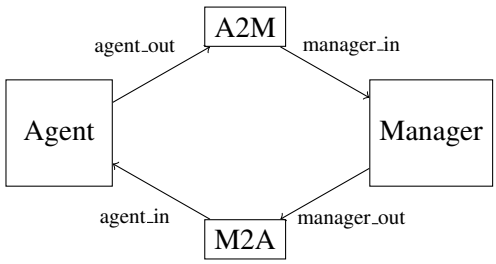


Figure 1: System layout

not pounds. This configuration is established during the session setup phase of the protocol. After a configuration has been established, we say that both the agent and the manager are *operating*.

3.1 Modelling the session setup protocol

To investigate the added value of formal modelling in a standardisation process, we constructed an initial model based on the 2008 version of the standard. We first describe the model that we constructed based on this version. In Section 3.2 we discuss the observations made during this initial stage.

Session setup is specified in the standard by two UML state charts [IEE08, Figures 10 and 11], one for the agent, one for the manager. A small part of the the agent state chart is shown in Figure 2. Multiple channels between agent and manager are supported by the standard. The systems must at least provide a primary, reliable channel. For such a channel it is assumed, among others, that messages are received in the order they are sent, and messages will not be duplicated or get lost. Session setup must take place along this primary channel, whereas all other communication is also allowed on this channel. We restrict our model to a single, primary channel between the agent and the manager modelled by two unidirectional buffers, `A2M` and `M2A`. Since we are mainly interested in the communication between agent and manager, and the correctness of the session setup protocol, we assume that all local triggers are synchronous, will be handled instantly, and require no time. A schematic view of the system we model and verify is given in Figure 1.

We systematically translate the state charts to mCRL2 using the translation from [HKL+09] to the processes `Agent` and `Manager`. We explain the translation in more detail using the state chart in Figure 2.

Example 1 *Each OR-state is modelled as a sort in mCRL2's data language. For the state chart in Figure 2, we obtain*

```
sort Agent_states            = struct Agent_Disconnected
                                    | Agent_Connected
                                    | ...
                                    | Agent_nop;
     Agent_Connected_states = struct Agent_Connected_Disassociating
                                    | Agent_Connected_Unassociated
                                    | Agent_Connected_Associated
                                    | ...
                                    | Agent_Connected_nop;
```

This declares the enumerated sort `Agent_states`*, with an element for the substate* `Connected`*, and* `Agent_Connected_states`*, with one element for each of the substates of the OR-state* `Connected`*.*

The element `Agent_Connected_nop` indicates that the state `Connected` is currently inactive. The agent process gets one parameter for each OR-state, which leads to the following declaration:

```
proc Agent(Agent_state: Agent_states,
           Agent_Connected_state: Agent_Connected_states, ...) = ...
```

This declares the process `Agent`, with parameters `Agent_state` and `Agent_Connected_state`, signifying the current state of the OR-state `Connected`. The initial state of the process is determined by the initial states in the state chart. If an OR-state is a substate of a state that is not active initially, it is assigned the corresponding `nop` value. For the example, we obtain the following initialisation.

```
init Agent(Agent_Disconnected, Agent_Connected_nop, ...);
```

The signals that are communicated between the agent and the manager, and the data that is communicated have to be sent along a single communications channel. Therefore, we combine both into a single data sort `Message`.

```
sort Message = struct signal(Signal) | data(Data);
```

This sort consists of elements `signal` which carry signals as parameters, and `data` which carries elements of sort `Data` as parameters. The sort `Signal` is built from the signals that are described in the state machine, for `Data` we use an abstract data type representing measurement data, consisting of two elements.

```
sort Data   = struct datum1 | datum2;
     Signal = struct sig_AssocAbort
                   | sig_AssocRelRsp
                   | sig_AssocRelReq
                   | ...
```

For readability of the model, we introduce functions that wrap signals into the `signal` constructor, producing a message. We write, e.g., the following, in which `AssocAbort` is a constant of sort `Message`, introduced by the keyword **`map`**. It is defined to be `signal(sig_AssocAbort)`, using the keyword **`eqn`**.

```
map AssocAbort: Message;
eqn AssocAbort = signal(sig_AssocAbort);
```

Messages are transmitted along the channels using actions `send` and `receive`, which communicate into the action `communicate`. In addition, actions are declared for local triggers, e.g. `assocRelReq`.

A transition can be taken, if the process is in the state from which the transition originates in the state chart and its trigger gets enabled. For local triggers, since we assume the trigger is processed instantly and atomically, we model this using a multi-action together with the trigger. The `assocRelReq` transition in Figure 2, is e.g. modelled as follows.

```
(Agent_state == Agent_Connected
   && Agent_Connected_state == Agent_Connected_Associated) ->
  assocRelReq|send(agent_out, AssocRelReq)
  . Agent(Agent_Connected_state = Agent_Connected_Disassociating, ...)
```

This says that, if the agent is connected and associated, upon triggering the local trigger `assocRelReq`, a signal `AssocRelReq` is sent immediately as defined by the `+entry` trigger in the `Disassociating` state. The state of the agent is updated according to the transition specification. In the Disassociating state, the transitions are modelled as follows.

```
(Agent_state == Agent_Connected
```

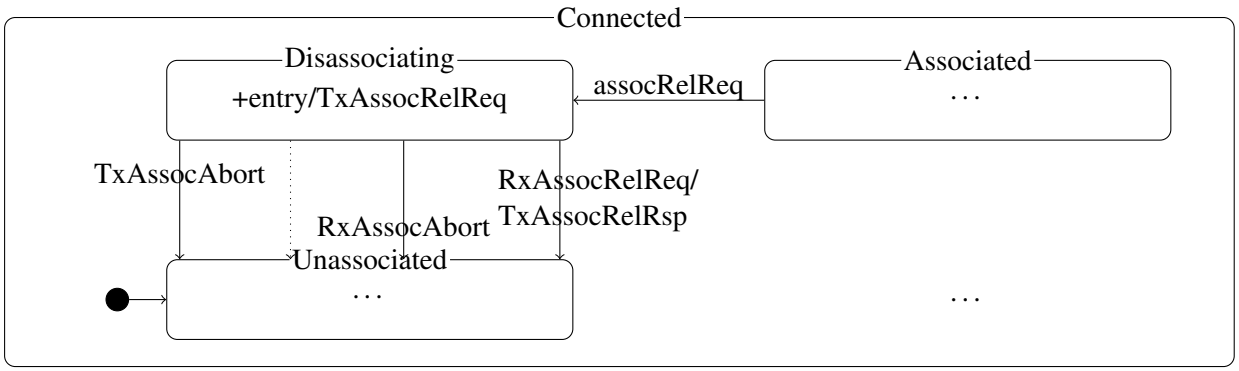


Figure 2: Excerpt of the UML state machine for the Agent

```
    && Agent_Connected_state == Agent_Connected_Disassociating) ->
  receive(agent_in(id), AssocRelReq)
  . send(agent_out(id), AssocRelRsp)
  . Agent(Agent_Connected_state = Agent_Connected_Unassociated)
+ receive(agent_in(id), AssocRelRsp)
  . Agent(Agent_Connected_state = Agent_Connected_Unassociated)
+ ...
```

The `+` models non-deterministic choice between two alternatives. The first alternative is receiving an `AssocRelReq`, which triggers sending an `AssocRelRsp`. The second alternative is receiving an `AssocRelRsp`, which does not trigger an action. In both cases, the system changes state to `Agent_Connected_Unassociated`.[2]

Note that the transition specifications in the state machines in [IEE08] are not complete. A full specification is given in the state transition tables in [IEE08, Annex E]. We combine the information from both sources. Although in general a single manager can serve multiple agents, for the purpose of this case study we restrict ourselves to a single agent and a single manager. This is the scenario that the standard focuses on; more specifically, the standard describes that the underlying communication layers are able to indicate that a connection between an agent and a manager has been established or aborted, hence the manager could serve each of the agents in their own context. According to the standard, both the Agent and the Manager can receive connect and disconnect indications from the transport layer at all times. We abstract from this behaviour, and assume that connect and disconnect indications are treated correctly, and we assume that always either both parties are connected, or both parties are disconnected. Upon disconnection, all messages still on the transport will be lost. This is a strong assumption on the capabilities of the transport layer, that allows us to focus on the session setup protocol.

The overall system is constructed using parallel composition of the agent and manager process, as well as the processes `A2M` and `M2A` representing the communication channels.

Example 2 *Sending of a message by the agent is modelled by a synchronous communication between the `Agent` and the `A2M` processes along channel `agent_out` as indicated in Figure 1, likewise for the other processes. This synchronisation is modelled using `comm` in the initialisation. Communication is enforced using the `allow` operator.*

[2] The full model is available at https://svn.win.tue.nl/viewvc/MCRL2/trunk/examples/industrial/ieee-11073/

```
init allow( {communicate, ... },
  comm( {send|receive -> communicate, ... },
    Agent(Agent_Disconnected, Agent_Connected_nop, ...)
    || A2M([]) || M2A([])
    || Manager(Manager_Disconnected, Manager_Connected_nop, ...))));
```

The standard describes the format used for configurations extensively [IEE08, Section 7.4]. In order to distinguish between different configurations, we use an abstract type for them. According to the standard, the agent provides the manager with a number of supported configurations in the association request. We do not provide these options, but assume that the agent supports all configurations of sort `Configuration`, and assume that the manager can choose a configuration non-deterministically. This non-deterministic choice of a configuration models the worst-case behaviour of any manager process.

Once session setup is complete, the protocol supports different types of data transmission. The manager is allowed to get and set values, and to invoke actions supported by the agent. The agent can send configuration updates and measurement data to the manager. Since we focus on the correctness of the session setup part of the protocol, we only include the transmission of measurement data in the operating phase. Transmission of measurements can be confirmed or unconfirmed, and in our model we only use the unconfirmed version, since this is the simplest part of the protocol in the operating phase, that still allows verification of the session setup.

For verification purposes, the agent and the manager can report their configuration through an action `operating(Configuration)` once the session setup protocol has finished.

3.2 Observations

While modelling the 2008 version of the standard, and checking deadlock freedom of our model during the process, we observed several flaws and omissions this version of the standard.

We were able to discuss the issues that we uncovered in this version of the standard with a member of the standards committee, who had access to a preliminary version of the 2010 version of the standard. This allows us to investigate how effectively problems can be found by formal modelling.

It turns out that some of the issues we discovered have been fixed independently by the committee in the 2010 version of the standard. These issues have been discovered over the course of two years that the standard was in use. Using our modelling approach, we have been able to find these issues in a few hours.

In the rest of this section we identify the different kinds of issues that we found in the 2008 version of the standard, and indicate occurrences of these problems. In case a problem is still present in the 2010 version of the standard [IEE10] we indicate so, otherwise the problem has been resolved by the standards committee.

Understandability We first discuss some general issues, that are not flaws in the the standard as such, but make the standard harder to understand. The main issue here is that the formalisms that are used throughout the standard, *i.e.* UML state machines, message sequence charts and the state transition tables are not properly introduced. This means that no clear semantics can be attached to the diagrams in the standard. An example of an unclarity is the "+entry/AssocRelReq" action in the Disassociating state of the agent [IEE08, Figure 11, page 60], see also Figure 2. It turns out that the `AssocRelReq` action is performed immediately upon entering the Disassociating state.

Since there is duplication of information between the state charts and the state transition tables,

it is important to understand which of the two is leading in case of an incompleteness or inconsistency. This should be clarified in the standard. Also note that the information in the state machines is incomplete. A model solely based on the state machines will lead to deadlocking behaviour.

In the rest of this section we investigate inconsistencies between different descriptions, and incompleteness of the descriptions in more detail.

Consistency If we take a closer look at the descriptions of the protocol in the state charts and the state transition tables, we observe several types of inconsistencies. First of all, inconsistent terminology is used to refer to the same thing in the state charts and state tables.

- The correspondence between the action names in the tables in [IEE08, Annex E], and the action names in the state charts depicted in [IEE08, Figures 10, 11] is not clear from the context. For example, the signals 2.8, 2.12 [IEE08, Table E. 2], represented by the events "aarq(*)" and "aare(*)" represent association requests and association responses respectively. These are represented in the state machines as `AssocReq` and `AssocRsp`.
- The triggers REQ agent supplied (un)supported configuration, in [IEE08, Table E. 3] (Signal ID 7.31 and 7.32) are not in the Manager state chart.

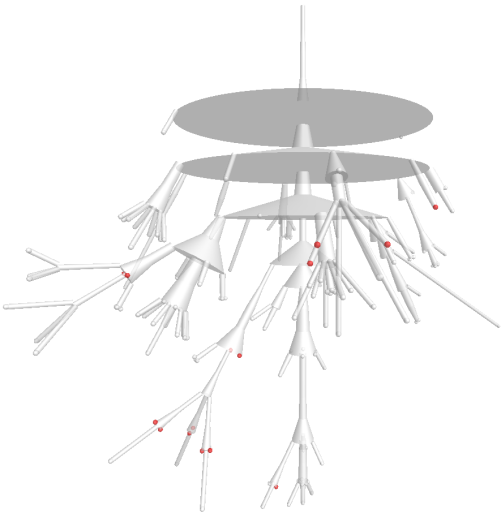
Second, the state transition tables and the state charts sometimes prescribe *different state changes* for the same event.

- According to the state chart [IEE08, Figure 11], the manager goes to the Unassociated state upon receiving a release request in the disassociating state, whereas according to Signal ID 9.16 in [IEE08, Table E. 3] it stays in the Disassociating state.
- The agent state chart [IEE08, Figure 10] demands that the connection is aborted immediately, whereas signal 9.16 in [IEE08, Table E. 3] mandates that the agent waits for its own release response. *This has not been fixed in the 2010 version of the standard.*
- According to [IEE08, Table E. 3], the Manager goes directly from the unassociated state to the associated state for signals with ID 2.9, 2.10 (and to Unassociated with 2.11). The state chart in [IEE08, Figure 11] contains an intermediate state Associating. After discussion it turns out that in this case the state chart is leading, and the table has been adjusted accordingly in the 2010 version of the standard.

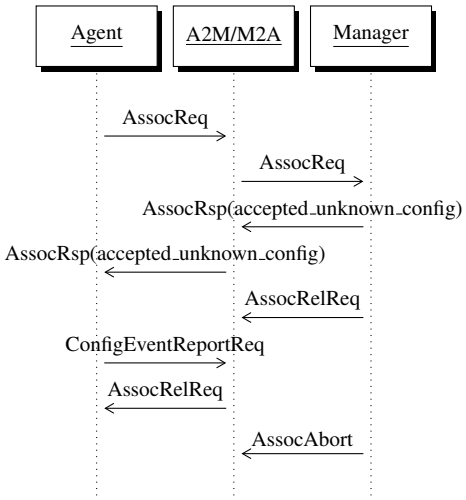
Completeness The last type of problem that we identified is related to completeness. It is sometimes not specified what should happen in case an unexpected message occurs. In principle, if we follow the description of UML state machines [Obj09], such a message can be ignored. However, since for most unexpected messages the behaviour is specified in the state transition table, we expect all such situations to be described. An example of this is the `Associating` state of the manager, for which [IEE08, Table E. 3] does not list any transitions.

All completeness issues were discovered through the deadlock checks that we carried out on our initial model. We describe the analysis in more detail in the rest of this section.

We generate the state space of the initial model, and, on the fly, check for deadlocks. For the initial model, a number of deadlocks are detected. Figure 3a shows an abstract visual representation of the state space of our initial model in the style of [vvv01]. The deadlock states are highlighted. One of the traces to a deadlock state is given in Figure 3b. For conciseness of the representation



(a) State space of an Agent and Manager process. Deadlocks are highlighted



(b) Trace to a deadlock state in the model of the 2008 version of the standard

Figure 3: Deadlocks

we have combined the two buffers into one in this picture. Inspection of this trace shows that it ends up in the situation where the Manager is in the `Unassociated` state, and the head of the input buffer of the Manager process is a `ConfigEventReportReq` message that has been sent by the Agent. The manager needs to process this message. According to the transition specification in the state machines and state tables, this message is not covered. In the rest of the standard the default procedure for treating unexpected messages is to accept the message, and respond with `AssocAbort`. Hence the obvious fix for these deadlocks is to *accept `ConfigEventReportReq`, and respond with an `AssocAbort` message*. Note that this message is not handled properly in a number of states, and we apply this fix in all places including the operating state.

If we explore the fixed model, we again run into deadlocks. This time, these are caused by the Agent process, which is not willing to accept `ConfigEventReportRsp` messages in the Unassociated state. The fix for this issue is similar to the one above. The model with the above two fixes is deadlock free.

4 Verification

In the previous section we have investigated the quality of the 2008 version of the standard. We have improved the model according to the findings. For more in-depth verification of the protocol we consider the 2010 version of the standard, in which we have fixed the remaining problems that we described in the previous section. We use this up-to-date version of the standard to ensure that our findings are most beneficial to the community using the standard.

The standard does not provide any clear requirements on the session setup phase. We formulate a number of requirements, which we subsequently verify. We express all properties in the modal μ-calculus, and check them on the model of the 2010 version of the standard, including the fixes from Section 3.[3]

[3] The requirements that we checked are available at https://svn.win.tue.nl/viewvc/MCRL2/trunk/examples/industrial/

1. The system can reach a state in which the manager successfully receives data from the agent.

$$\langle true^* \rangle \quad \langle \exists_d . communicate(manager\_in, data(d)) \rangle true$$

2. From every state, the system can reach a state in which the manager can successfully receive data from the agent infinitely often, within a single session.

$$[true^*] \quad \langle true^* \rangle \nu X . \langle (\neg \exists_{id} communicate(agent\_out, AssocReq(id)))^* \rangle \\ \langle \exists_d communicate(manager\_in, data(d)) \rangle X$$

3. When both the agent and the manager reach the operating state, then they agree on the configuration that is being used.

$$[true^*] \quad \forall_{c,c'} [operating(c) | operating(c')](c = c')$$

4. Before measurement data is correctly communicated using the protocol, it first needs to be ensured that the agent and manager agree on the configuration that is used.

$$[true^*] \quad \forall_{c,c'} [operating(c) | operating(c')] \\ [(\neg \exists_{ch,m} m \in \{AssocAbort, AssocRelReq, AssocRelRsp\} \wedge communicate(ch, m))^*] \\ [\exists_d . communicate(manager\_in, data(d))](c = c')$$

Requirements 1 and 2 serve as a check to see that data can be communicated as expected. Both properties hold for our model of the 2010 version of the standard. The first requirement ensures that requirement 4 does not hold trivially.

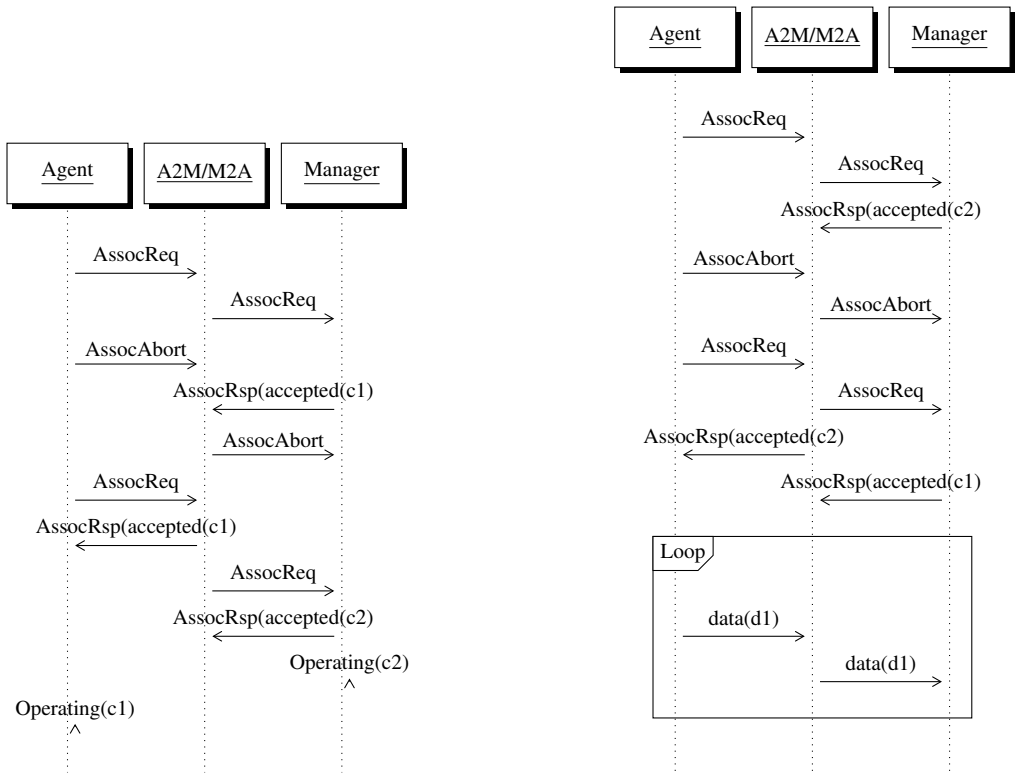
Our main focus is on requirements 3 and 4, *i.e.* determine whether an agent and a manager can get into an operational state with different assumptions on the data that is communicated, *e.g.* a state in which the agent thinks weights are transferred in kilogrammes, whereas the manager assumes weights are in pounds.

Requirement 3 is the requirement on consistent operating states as we originally formulated it. This property does not hold for the system. In fact, this is not a problem, as long as no data is successfully received by the manager in inconsistent operating states. This weaker requirement is expressed in requirement 4. We investigate both properties in detail in the rest of this section.

4.1 Reachability of inconsistent operating states

If both the agent and the manager are in the operating state, they are ready to transmit measurement data. Since measurement data is interpreted according to a configuration, configurations should be consistent between the agent and the manager.

Given that the agent and the manager report their configuration if they are in the operating state, this problem can be formulated as a reachability problem. The property holds if, in all situations in which a multi-action `operating(c)|operating(c')` can be reached, it holds that $c = c'$. If we verify this property for the system, we find that the property does not hold. A counterexample trace to a state with



(a) Trace to operating states with inconsistent configurations

(b) Trace to state with inconsistent configurations with data transfer

Figure 4: Counterexamples

inconsistent operating states is shown in Figure 4a. If we carefully inspect the counterexample that is produced, observe that the action `operating(c2)|operating(c1)` happens in a situation where the head of the input buffer of the manager contains an `AssocReq` message, and the head of the input buffer of the agent contains an `AssocRsp(accepted(c2))` message. Reception of either message, will cause the sending of an `AssocAbort` message by the receiving party. Since on any computation path, one of these messages will be processed, before measurement data is processed, the session will always be aborted. Furthermore, measurement data is not accepted in the Unassociated state, in which both processes end up after processing the first message in each of the buffers.

Intuitively, it is clear that this concrete counterexample does not cause any safety problems. This gives rise to the weaker property that is verified in the following section.

4.2 Data shall not be transmitted in inconsistent operating states

Requirement 4 weakens requirement 3 by allowing the agent and the manager to reach inconsistent operating states, yet requiring that measurements are only successfully received by the manager if the operating states are consistent.

The structure of the formulae are similar, yet in this case, after the operating states have been reported, we allow arbitrary actions, except those that can move a system away from its operating

ieee-11073 in their concrete syntax.

state. Then, if the manager can successfully receive data, the configurations of the agent and the manager are required to be the same. If we verify this property using the mCRL2 toolset, we find that the property does not hold. Inspection of the model gives us the counterexample depicted in Figure 4b.

The counterexample is, to a large extent, similar to the counterexample we saw previously. A session is set up, but the setup process is aborted by the agent before it has received the association response from the manager. The agent then initiates a new session setup, in which it interprets the association response from the first setup as the response corresponding to the second setup. At the point in which the `operating(c2)|operating(c1)` actions happen, the input buffer of the manager is empty, and the input buffer of the agent contains a message `AssocRsp(accepted(c1))`. Since the agent is in the operating state, it can send data, as long as it does not read this message from the buffer. Furthermore, since the manager has accepted the configuration, it is also in the operating state, hence it can accept the data the agent has sent.

5 Fixing the protocol

The property violations that we have seen in the previous section are caused by the mixing of `AssocRsp` messages across different sessions. We propose fixing this issue by adding a session identifier to the `AssocReq` and `AssocRsp` messages. Note that duplication of session identifiers is not allowed.

The changes that we make are as follows. The agent and the manager keep track of the largest session id `t` they have seen up to that point, initially we assume the session id is zero. In the `AssocReq` message, the agent includes `t+1`. Upon reception of this message, the manager updates her session id to the value included in this message, and includes the session id in the `AssocRsp` message. If the agent receives an `AssocRsp` message with a session id that is not equal to the session id that it stored, it will discard the response, send an `AssocAbort` message, and move to the unassociated state. This is similar to the treatment of other unexpected messages in the protocol.

It is crucial that session identifiers are unique and may not be repeated. In case duplicates are allowed, the agent could repeatedly send `AssocReq` and `AssocAbort` messages, until a duplicate session id is reached, and then read an old association response from the input buffer, ending up in the same problematic situation as in Figure 4b.

The state space of the modified system is infinite. Since session identifiers need to be unique, there is no straightforward way of doing the verification in an exhaustive way. Instead we verify a version of the model in which we count session identifiers modulo some maximum value `maxSessionId`, and, once the first session identifier is reused we clear the buffers. Basically, this resembles the fairness assumption that, once the agent has attempted `maxSessionId` session setups, it has also read `M` messages from its input buffer, where `M` is buffer size. For this version of the model we are able to verify that all properties hold. Given the correct behaviour of the protocol in the case with periodic session identifiers and fairness assumptions, we argue that it is reasonable to assume that the protocol is also safe in the case without fairness assumptions if session identifiers are never reused.

6 Conclusions

In this paper we investigated the IEEE-10073-20601 standard for personal health devices. An analysis of the 2008 version of the standard uncovered several omissions and inconsistencies. The omissions in the standard introduce deadlocks in the system, but these can easily be fixed. Most of the issues uncovered have been fixed in the 2010 version of the standard. One issue remains unfixed.

A further analysis of the 2010 version of the standard, including a fix for the remaining problem,

revealed that agents and manager can reach inconsistent operating states. This is problematic, since measurement data can be transferred from the agent to the manager, and be interpreted incorrectly by the manager. Use of the incorrectly interpreted measurement data poses a serious health hazard.

We fixed the protocol by introducing a session identifier, and verified that the fixed version of the protocol satisfies the requirements. Given the bugs we detected, we recommend the full verification of the rest of standard.

By means of the case study performed in this paper, we demonstrate once more that formal modelling is an invaluable addition to the standardisation process. The main lessons that we learned during this case study are the following:

- *Omissions and inconsistencies in the standards document cause easy to avoid bugs.* Formal modelling of the standard allows the detection of these problems in a short period of time, whereas it took the industry two years to find the same problems.
- Requirements are typically missing from standards documents. As a result, the purpose of specific parts of the standard is not clear. Furthermore there is no way of determining whether the standard is correct, due to the absence of a clear verification question. To improve this, *standards should provide clear requirements.*
- *Protocol bugs in the standard cannot be fixed without breaking standard compliance of existing devices.* Serious bugs in existing protocol implementations are likely to stay around.
- Without formal verification, *subtle bugs end up in published standards.* As a result, bugs also find their way into devices, which can lead to dangerous situations for their users.

It is our conviction that any unverified communication scheme is likely to contain bugs. We have demonstrated once more that formal modelling is an invaluable addition to the standardisation processes. If requirements are explicitly formulated in standards documents, formal verification can be used to verify the protocol that is described. Bugs that are not found in an early stage of the development of the protocol are likely to stay around in devices once they hit the market, resulting in risks for the users of these devices.

We are lead to believe that formal verification should be considered as an integral part of the development process of communication standards.

Acknowledgements: We are grateful to an anonymous member of the standards committee, for his support and fruitful collaboration in this preliminary analysis. Furthermore, we would like to thank Erik de Vink and Tim Willemse for their helpful comments on an earlier version of this paper. Finally, we would like to thank an anonymous reviewer, whose comments helped improving the presentation of the lessons learned.

Bibliography

[BB01] B. Boehm, V. Basili. Top 10 list [software development]. *Computer* 34(1):135 –137, jan 2001.

[BBR10] J. C. M. Baeten, T. Basten, M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science 50. Cambridge University Press, 2010.

[BK84] J. A. Bergstra, J. W. Klop. Process algebra for synchronous communication. *Information and Control* 60(1-3):109–137, 1984.

[DFH+05] M. Duflot, L. Fribourg, T. Herault, R. Lassaigne, F. Magniette, S. Messika, S. Peyronnet, C. Picaronny. Probabilistic Model Checking of the CSMA/CD Protocol Using PRISM and APMC. In *Proc. AVoCS 2004*. ENTCS 128(6), pp. 195 – 214. 2005.

[DGRV00] M. Devillers, D. Griffioen, J. Romijn, F. Vaandrager. Verification of a Leader Election Protocol: Formal Methods Applied to IEEE 1394. *Formal Methods in System Design* 16:307–320, 2000.

[Fru06] M. Fruth. Probabilistic Model Checking of Contention Resolution in the IEEE 802.15.4 Low-Rate Wireless Personal Area Network Protocol. In *Proc. ISOLA'06*. 2006.

[GM99] J. Groote, R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In Haeberer (ed.), *Algebraic Methodology and Software Technology*. LNCS 1548, pp. 74–90. Springer, 1999.

[GMR+08] J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, M. van Weerdenburg. Analysis of distributed systems with mCRL2. In Gardner (ed.), *Process Algebra for Parallel and Distributed Processing*. Pp. 99–128. CRC Press, 2008.

[GPW03] J. F. Groote, J. Pang, A. G. Wouters. Analysis of a distributed system for lifting trucks. *JLAP* 55(1-2):21 – 56, 2003.

[GW05] J. F. Groote, T. A. C. Willemse. Parameterised boolean equation systems. *TCS* 343(3):332–369, Oct. 2005.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.* 8:231–274, June 1987.

[HKL+09] H. H. Hansen, J. Ketema, B. Luttik, M. Mousavi, J. van de Pol. Towards Model Checking Executable UML Specifications in mCRL2. In *Proceedings of the 2nd IEEE International workshop UML and Formal Methods (UML&FM 2009)*. 2009.

[IEE08] IEEE. Health Informatics-Personal Health Device Communication Part 20601: Application Profile- Optimized Exchange Protocol. 12 2008.

[IEE10] IEEE. Health Informatics-Personal Health Device Communication Part 20601: Application Profile- Optimized Exchange Protocol. 5 2010.

[MG05] A. J. Mooij, N. Goga. Dealing with Non-local Choice in IEEE 1073.2s Standard for Remote Control. In Amyot and Williams (eds.), *System Analysis and Modeling*. LNCS 3319, pp. 257–270. Springer, 2005.

[MGWB03] A. J. Mooij, N. Goga, W. Wesselink, D. Bosnacki. An Analysis of Medical Device Communication Standard IEEE 1073.2. In *Proc. 2nd IASTED Int. Conf. on Communication Systems and Networks*. ACTA Press, 2003.

[MP07] A. Mathijssen, A. Pretorius. Verified Design of an Automated Parking Garage. In Brim et al. (eds.), *Formal Methods: Applications and Technology*. LNCS 4346, pp. 165–180. Springer, 2007.

[Obj09] Object Management Group. OMG Unified Modelling Language Superstructure Version 2.2. Feb. 2009. Accessed 1 June 2012.

[Pet62] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, 1962.

[Rom99] J. Romijn. *Analysing Industrial Protocols with Formal Methods*. PhD thesis, 1999.

[SR09] F. P. M. Stappers, M. A. Reniers. Verification of safety requirements for program code using data abstraction. *ECEASST* 23, 2009.

[vRG03] I. van Langevelde, J. Romijn, N. Goga. Founding FireWire Bridges through Promela Prototyping. In *Proc. 17th Int. Symposium on Parallel and Distributed Processing*. IPDPS '03, pp. 239.1–. IEEE Computer Society, Washington, DC, USA, 2003.

[vvv01] F. v. Ham, H. v. d. Wetering, J. v. Wijk. Visualization of State Transition Graphs. In *Proc. IEEE Symp. Information Visualization 2001*. Pp. 59–66. IEEE CS Press, 2001.